Bachelor thesis
Computer Science

Radboud University Nijmegen

MapReduce Framework Performance Comparison

Author:
Thomas Nägele
t.nagele@student.ru.nl
4031253

First supervisor/assessor:
prof. dr. F.W. Vaandrager
F.Vaandrager@cs.ru.nl

2nd July 2013

Abstract

This paper describes my effort to verify the speedup results of the Phoenix 2 framework. This framework is an implementation of MapReduce, which is a programming model for distributed computing without having the programmer to write parallel code. Reproducing the speedup results fails partly, due to performance issues in the Phoenix 2 implementation. In addition the achieved speedups for Phoenix are compared to those of Hadoop, another large MapReduce framework, which is very popular amongst large companies. Hadoop seems to scale better compared to Phoenix, but an explanation for these results might be the performance issues of Phoenix. In addition, the number of tests that is used for comparison is too low to draw a solid conclusion.

Contents

Chapter 1

Introduction

1.1 Problem statement

MapReduce [2] is a programming model invented by researchers at Google. It is meant for processing large amounts of data by a large cluster. Since it is hard to parallelize such large amounts of data over multiple hundreds of machines, the authors developed the MapReduce model. The MapReduce algorithm is easy to use and handles all parallelization, so the programmer does not have to concern about that. Google used the MapReduce model for many applications, such as the indexing of websites on the internet. They use large clusters of cheap machines to process all data.

Since MapReduce was originally developed for high performance computations on heterogeneous clusters of computers, there are quite some research papers about optimizing performance of multi-core computers. Problems may occur in these configurations because of the processors having one shared memory instead of having smaller pieces of memory available for each single processor core. One of those optimized implementations of MapReduce is the Phoenix System [9]. The Phoenix System was originally published in the April of 2007 [3] with a follow up a few years later, called Phoenix 2. This follow up included support for Linux x86_64 systems and was more stable than the original. In the year 2011 a reimplementation of Phoenix 2 was created, written in a different language: C++. This reimplementation was called Phoenix++ [4] and was released together with the publishing of an article.

There are of course many more implementations of MapReduce. One of the biggest frameworks for MapReduce is Hadoop [5]. Hadoop is used by large, international companies like Amazon and Facebook [6]. Hadoop uses its own file system (HDFS) as storage for its machines and can be used for many purposes.

This study may be split into two main goals. These goals are as follows.

1. To verify the speedup results as stated in the original paper about Phoenix [3] by using Phoenix 2 on two different machines.

2. To compare the Phoenix 2 framework to the Hadoop framework in terms of speedup.

The first chapter is this introduction. The second chapter contains some more information about the MapReduce model. The third chapter will cover the first goal while the fourth chapter covers the second goal. The fifth chapter gives a brief overview of the related work that is used for this research.

1.2 Motivation

One of the most convincing arguments for me to chose this research area is because I have always liked High Performance Computing. Because of the simplicity and power of the algorithm it has found many useful purposes in daily routines run at Google. Because of its diversity, much research on the algorithm is done and many optimizations are developed. All this research about optimizations and different applications of the algorithm concludes with a demonstration about how fast the algorithm is and why it is better then other solutions for the same problem. Only few of those demonstrations are actually re-run on other computers or in other environments in order to verify these results. Because this verification step is ignored in many different researches, I would like to challenge one of the implementations for a test. Hereafter, I'd like to compare these results to a larger and widely used framework: Hadoop. The main focus of the research will be the speedup in general, because the scalability is expressed in this way, which is one of the most important features of the MapReduce algorithm.

Chapter 2

MapReduce

First - and most important - of all, one must understand what MapReduce does and why it is more efficient than writing native multi-threaded code for different applications. MapReduce is an algorithm that can be used for many applications just by writing one library and simply implementing a few required functions to run MapReduce on your own data set. MapReduce can do a lot of things, if the user is able to define both a Map and an Reduce function for the application. MapReduce was originally meant for large distributed systems, containing a few hundred physical machines. These machines are connected to each other by a network that may be global (internet) or local (intranet). In the following paragraphs more information about the components of MapReduce will be provided.

2.1 First application

The machines that were used at first were pretty standard personal computers, containing two processor cores and 2 to 4 GB of RAM [2]. They also contained normal IDE hard drives, since they are very cheap and fast enough for the operations that must be performed. These machines are all linked together through ethernet interfaces with speeds of 100 Mbit or 1 Gbit. Nowadays clusters like this are still very cheap and efficient ways of having a large computation cluster.

2.2 A cluster

A cluster of computers is split into nodes. Typically each processor or physical thread will be one node. A whole network contains one Master node. This Master node keeps an overview of all other nodes. All the other nodes are worker nodes. Tasks will be assigned to them and they will execute. All the work that needs to be done is split into a number of tasks. In MapReduce a task will be a map task or a reduce task.

2.3 Map

The Mapper is the function that processes the data by reading all the data and returning a list of key-value pairs. Each pair contains the object that is processed and the value that is sent to the reducer. These pairs will be noted as following:

<key , value>

The results of the map function are stored in the memory of the machine. Periodically these values are written to the disk of the machine and the Master node is notified of the location of the intermediate results. The Master node then marks this mapper node as free and may assign a new task to it. This task may either be a map task or a reduce task.

For example: When all the words in one document are counted by MapReduce, the Map function just runs through the document and returns a pair for each word it encounters. The key is set to the word that is just read and the value is set to 1. All these pairs are stored in the local memory and finally written to the disk. For example: if the text 'more and more' is read, the following pairs are stored:

<more,1>
<and,1>
<more,1>

Notice that the same pair `<more,1>` is stored twice.

2.4 Reduce

The Reducer is the function that reads the intermediate data from the Mapper and reduces this by processing the objects by running the user-defined Reduce function. First it receives a location of the mapped data from the Master node and fetches this data. Then this data must be ordered by key. Then the Reducer iterates through the data and groups all keys that are equal by processing their values. Finally this output file is written and the Master node is notified.

If the output of the above Mapper is processed by a Reducer, the following file will be generated.

<and,1>
<more,2>

You can see that the key 'more' is grouped and their values are added up, while the key 'and' did not change at all. Also the output is now ordered by key.

The output files of the Reducer tasks may be input for another MapReduce run or may be used as seperate files.

2.5 Master node

When a MapReduce task is started, one Master node is chosen from all machines in the network. This Master node is the machine that delegates tasks to other machines and is also the only node that has a complete overview of what is happening in the network. The Master node assigns new tasks to worker nodes and reassigns tasks that take too long. A sequence overview is as follows.

1. The input data is split into a number of pieces of a specified size. The algorithm is started on all nodes.
2. One node is set to be Master node and starts delegating work to other nodes. All pieces created in the first step are first mapped by the mapping function. The number of reduce tasks at the start should be low.
3. If a worker gets a map task, it runs the map task and stores the result in the memory of the machine.
4. Periodically these stored results are written to the disk and the Master node is notified of the location.
5. When the Master node gets notified about a location of mapped pairs, it will start a reduce tast on one of the free workers.
6. When a reduce task is called, first of all it fetches the stored results from the remote machine on which the map task has run. Secondly, these results are sorted by key. Thirdly, the results are reduced.
7. When there are no more data to process, the Master node returns the final results to the user program.

All this time the Master node has an overview of what all nodes are doing. The master will also re-assign already assigned tasks to idle nodes, because this might improve overall performance.

2.6 Benefits

One of the main benefits of using MapReduce is that there is only one library required for a large number of different applications. Once the MapReduce library itself is complete, it is easy to write applications such as word_count (that counts words in documents) or other large data processing applications for MapReduce. For each application only the Map and Reduce functions must be defined according to the basic idea of MapReduce. MapReduce itself takes care of all parallelization and machine failures. This means that the Mapper should return results as a list of `<key,value>`-pairs and the Reducer must be able to read them. Because of the abstract implementation of MapReduce, it is possible to let the algorithm compute many different things with relatively small adjustments to the code. Because of this main benefit, the size of the code can be smaller when developing an application, as shown by Standford University[9].

MapReduce is designed to handle errors in execution. The Master node checks regularly whether a worker is still active or not. This can be done by pinging the workers. When the Master node detects a problem with one of the workers, it will re-assign the task that the worker was doing to another worker.

When the moment has come that there are no more new tasks to assign, the Master node will assign tasks that are already being executed by other workers to idle workers. The benefit of this process is that you will loose less time when one worker fails. Besides this benefit, there is a chance that a worker who has got a task as back-up, finishes the task sooner than the worker who got the assignment first. The result of this speedup is marginal, especially when the computers in the network are equally fast.

Chapter 3

The Phoenix System

3.1 Preliminaries

The Phoenix System (hereafter called Phoenix) [3][9] is an implementation of MapReduce by Stanford University. It is made for shared-memory systems, instead of large, distributed clusters of computers. It only uses the threads that are available on one single computer. Phoenix consists of an API that can be used by developers to write programs for.

3.1.1 The Phoenix API

The Phoenix API is the interface of Phoenix that was originally written in C, but with version 2.0 an implementation for C++ was added. The programmer only has to create a few simple functions (e.g. Map and Reduce) and the library will handle the threading and load balancing. Figure 3.1 shows the model of the data flow in the Phoenix API.

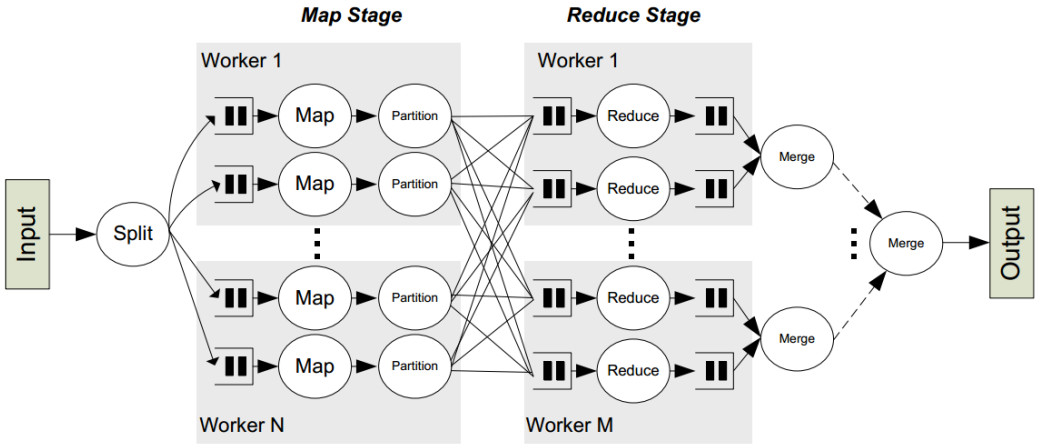

Figure 3.1: The Phoenix data flow model.

There are three functions that must be defined by the user and two functions that are optional. The required functions are Split, Map and Compare. The last function is required, because it is necessary to know how to compare two keys before reducing or merging. The other required functions speak for themself. The Reduce and Partition functions are optional, because the system will use default functions for them instead if they are not defined.

3.1.2 Speedup

In the original article about Phoenix [3], the authors stated some speedups that were achieved by using Phoenix. All the tests were run on two different systems and the speedup was calculated. These speedups, for different numbers of threads and different tests, are displayed in figure 3.2 and 3.3.

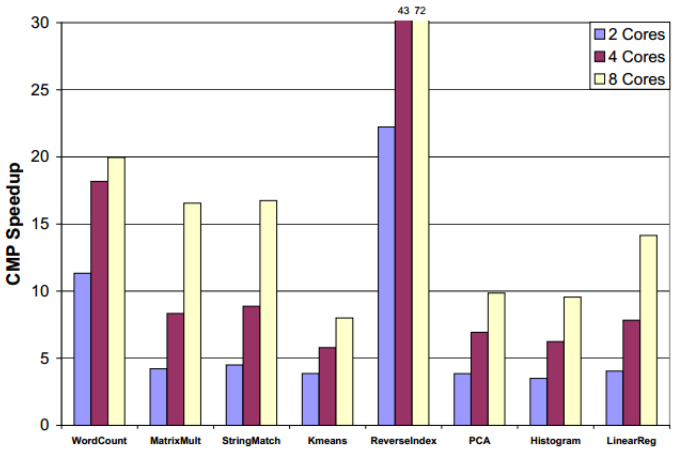

Figure 3.2: Phoenix speedups achieved on the CMP system.

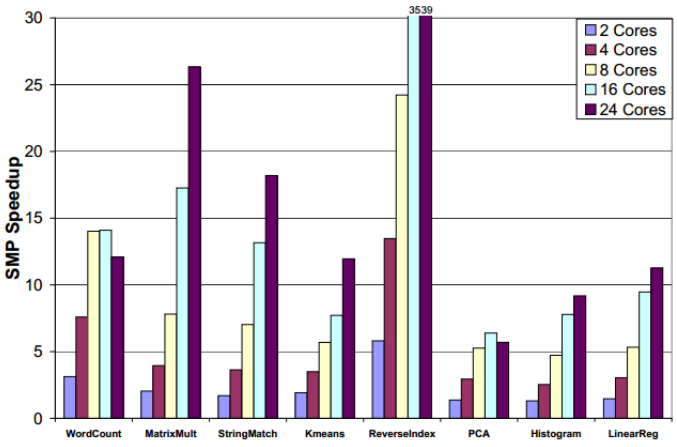

Figure 3.3: Phoenix speedups achieved on the SMP system.

The superlinear speedups when running tests with less than eight threads are probably due to caching within the CPU, so less speed is lost on storing intermediate results. With more threads, more results need to be stored, which is expensive, resulting in a lower speed increase, or even a decrease. This part will not be covered by this research.

3.2 Validation of experiments

One goal of this research is to implement and test the Phoenix implementation of MapReduce. The main target is to verify the achieved speedups as stated in the article about Phoenix[3] in figure 2. In this section, the method to do so is given, as are the results of these tests.

The Phoenix implementation of MapReduce is available for download from the website [9] and may be used freely. The framework runs from version 2.0 also on Linux x86_64 systems, which are more commonly used than Solaris OS. The sources must be compiled, after which the basic functionality of the framework should be working. The package that is downloaded includes some tests, that are used for their article. Also the used data sets are available for download, in order to enable others to redo the experiment or use the framework.

3.2.1 Research plan

Description

After having installed and configured the Phoenix system, the tests that are included must be run. Like in the original article, all tests of all data set sizes are exeuted multiple times on different numbers of threads. Depending on the number of threads that the system can handle, there are different numbers of tests that should be ran. For every 2^t threads a test of 10 runs is executed for every data set. Obviously the maximum number of threads that can be executed on one system is the number of threads the processor can handle in parallel. For example a system with a processor containing 8 logical threads can execute tests for $2^0 ... 2^3$ threads, because 2^3 is the number of logical threads available on the system.

Tests

For this paper, 3 tests are executed. Brief descriptions are taken from the original article[3].

- **Word Count**
  Counts words in the input document and returns the most occurring words.
  Sizes: 10, 50, 100 MB.

- **Histogram**
  Counts all the occurring RGB colors (separately) in a given bitmap image.
  Sizes: 6816 x 5112, 13632 x 10224, 25000 x 18750 pixels (respectively 105, 418 and 1400 MB).

- **Linear Regression**
  Computes a line that best fits a file containg coordinates.
  Sizes: 50, 100, 500 MB.

These three tests have 3 different data sets each. One small (S), medium (M) and large (L) data set. Hereafter, these data sets will be referred to by their size letter with the name of the test.

All these tests are executed by both MapReduce and the sequential algorithm that is provided. Because the sequential algorithm only uses one thread, this test should run only once on each data set size. The resulting speeds of this sequential algorithm will be used for calculating the speedups of the MapReduce versions.

Systems

There are two different systems that will run these tests. One system is a simple notebook pc with an i7 CPU and the other is a large server containing a large Opteron CPU.

	NBP	LOS
CPU	Intel Core i7 3610QM	AMD Opteron 6276
CPU Count	1	4
Core Count	4	16
Threads/Core	2	1
L1 Cache	4 x 32 KB	16 x 16 KB 4-way
L2 Cache	4 x 256 KB	8 x 2 MB 16-way
L3 Cache	6 MB	2 x 8 MB max 64-way
RAM Memory	16 GB	128 GB
OS	Ubuntu 13.04	Ubuntu 12.04 LTS

These two systems are not much alike, because one of the machines is a server and the other is a simple multimedia notebook. Howerver, they both are shared-memory systems, so they meet the requirements for the Phoenix system.

Speedup measurement

To be able to measure the speedup that MapReduce can give compared to the sequential implementation of the test, both the sequential test and the MapReduce are executed. Every test will be run 10 times, so that an reliable average can be taken. When all tests have been executed, each number of threads has an average time for each size of dataset that is used in the tests, plus one time for the sequential run of the algorithm. The speedup for every MapReduce setting can than be calculated by dividing the sequential execution time by the execution time for the MapReduce algorithm for the same data set (size). The speedup may be expressed by the following formula.

$$S = \frac{t_s}{t_m} \tag{3.1}$$

Where S is the speedup, t_m the execution time with the MapReduce algorithm and t_s the sequential execution time. In this formula, these execution times may only be compared to each other if the used data set and the size of this data set are equal to each other.

3.2.2 Research in practice

In this section, an overview of the experiments, adjustments and problems that occurred is given.

Code adjustments

In order to be able to run sets of tests efficiently and without supervision, the code of the Phoenix tests must be modified. Instead of output about what the algorithm is doing at the moment, just timing information is required. In this case, the only time that is needed, is the time it took to run the algorithm for one run on a particular data set. There are predefined methods to handle timing within the program, but they are set up in a way that the different phases of MapReduce are timed, instead of the whole program. Also these functions are monitoring the lapsed time, but only print the full seconds of it. Because computers are fast nowadays and little differences in timing could matter, this output function needed to be modified, so that the lapsed time in seconds, including the microseconds is displayed to the user. Therefor the following rule was added to every test that was ran.

```
double spendtime = ((endtime.tv_sec+endtime.tv_usec/1000000.0) -
    (starttime.tv_sec+starttime.tv_usec/1000000.0));
```

In this line, the integer values of the microseconds are cast to doubles and added to the seconds, which gives doubles. Next the *starttime* must be substracted from the *endtime*.

The *starttime* is set just before initializing the MapReduce algorithm, but after settings the algorithm parameters. The *endtime* is set after releasing all variables that are stored in memory. Then the time that is spent is calculated and printed to the terminal or file.

Besides this, all other text that is normaly printed to the terminal is commented out, so that only the lapsed time is displayed and these values can easily be stored into a simple CSV file. In this way, the measured times are somewhat easier to analyze.

These code adjustments are made to all three tests that are distributed with Phoenix in both the MapReduce version of the program as well as the sequential file. The execution flow for the MapReduce functions is given in the *Test code abstract* [A.1].

Running batches

To make life easier, running batches of benchmarks is useful. Therefor a bash script is used for running multiple tests in a row. To run one test with one specific configuration, the bash script *runBatch.sh* [A.2] is used. The parameters for this script are as follows.

Parameter	**Value**
$1	The program to be executed
$2	The data set to analyze
$3	The number of runs
$4	The number of threads (for MapReduce)
$5	The cache size (for MapReduce, in bytes)

3.2.3 Results

Figure 3.4 presents the average speedups that are measured after running 10 cycles of every test on every possible size of data set. The speedups are computed for 2, 4 and 8 threads.

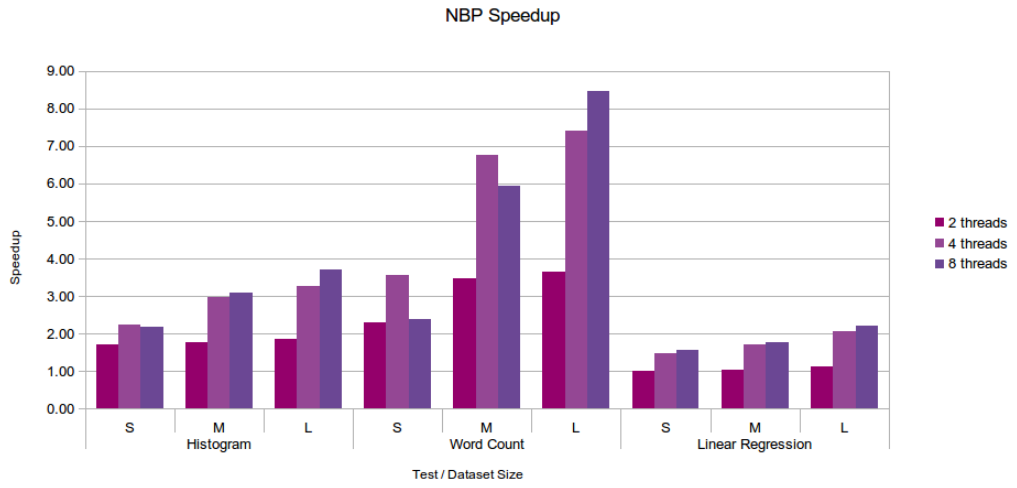

Figure 3.4: Phoenix speedups achieved on the NBP system.

Figure 3.5 presents the same speedup measurements for the LOS system. All tests were run on 2, 4, 8, 16, 32 and 64 threads, since this system has a maximum of 64 logical threads.

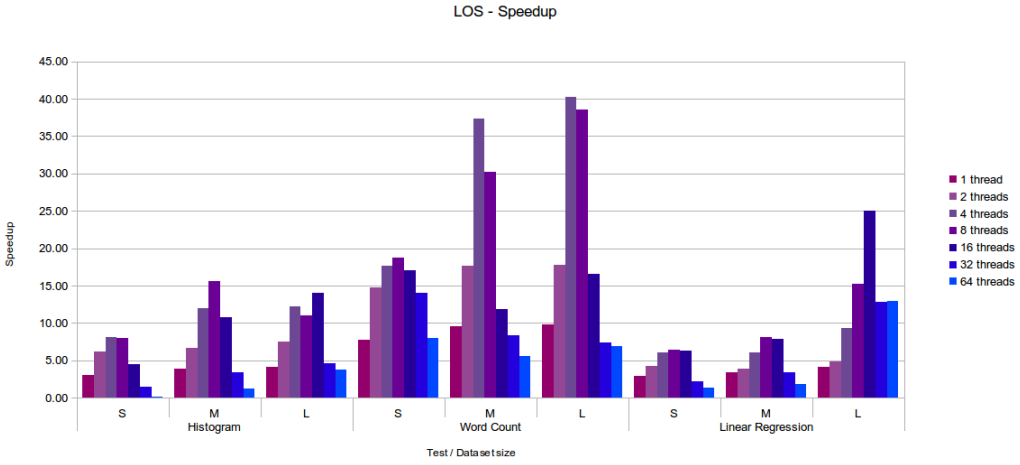

Figure 3.5: Phoenix speedups achieved on the LOS system.

3.2.4 Performance issues

While running all the tests, some issues occurred. Two of these issues forced the research to be executed in a slightly different way than planned. One of the issues has to do with the idling of the hard drive, while the other one had to do with the chunk size of the pieces in which the input data are split.

Hard drive idling

The first problem appeared when fetching data from a conventional hard drive, the hard drive must first start spinning, if the drive was in idle state before. The time it takes to find the requested data file increases significantly if this happens. This was only an issue if the drive was in idle state, so only the first few results of each batch of tests may be significantly slower than their follow-ups. In some cases, the difference of the first three runs was so different from the next seven, that these runs can be better left out. To be on the safe side, I've decided to ignore the first five runs of each series. Because this will only leave five measurements for further analysis, all tests must be re-run. The idea is to execute 15 runs of every test with every data set size, and ignore the first 5 runs in the analysis.
Figure 3.6 presents the speedups that were achieved on the NBP system with this method.

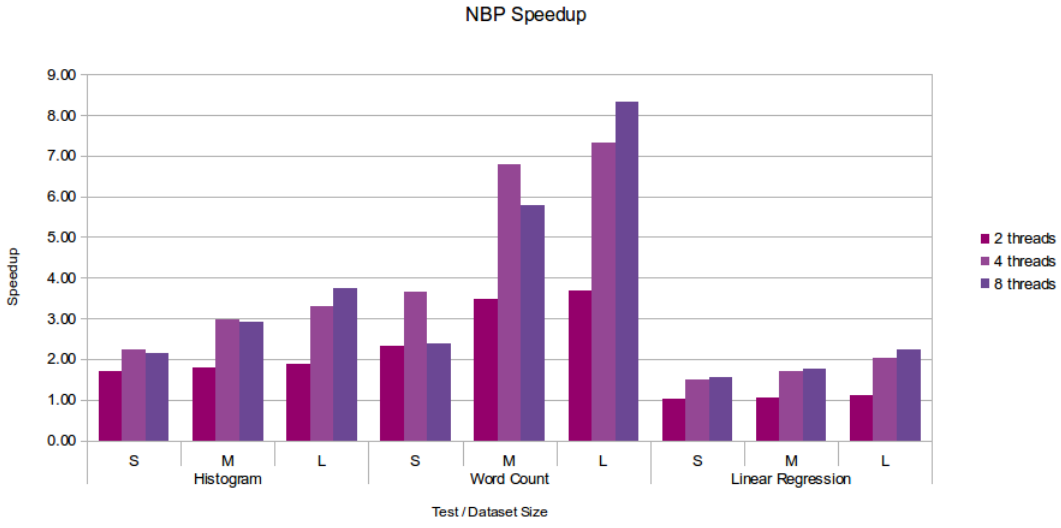

Figure 3.6: Phoenix speedups achieved on the NBP system, based on the last 10 measurements of 15 runs.

The only noticeable difference between the two batches is that the Histogram M data set running on 8 threads, is now somewhat slower than the same set on 4 threads, while running the test only 10 times, the speedup increases slightly. The difference however is very small and barely noticeable. Figure 3.7 presents the speedups, based on the last 10 runs out of 15 on the LOS system.

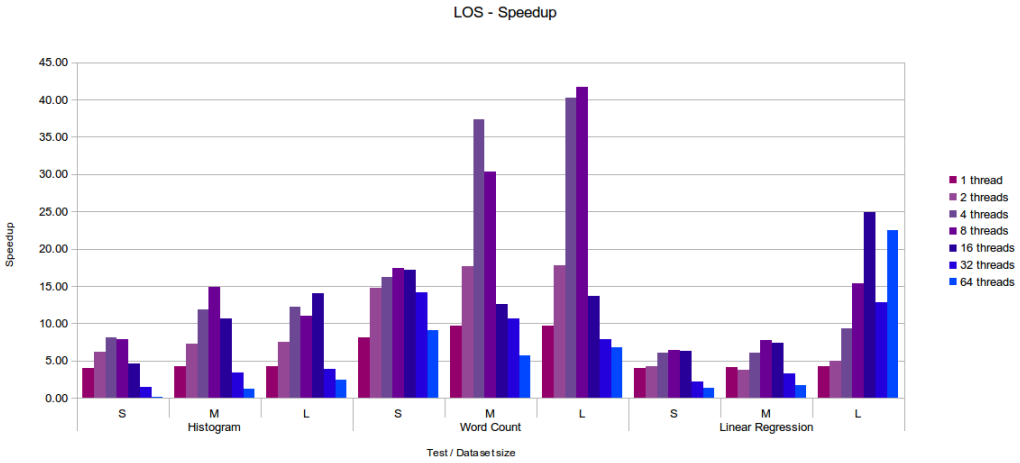

Figure 3.7: Phoenix speedups achieved on the LOS system, based on the last 10 measurements of 15 runs.

Also in these measurements, there are not much noticeable differences in the outcome. The biggest difference is the speedup that is achieved in the large linear regression test, running on 64 threads. While this test performed the same as on 32 threads when running 10 tests, the test completes a lot faster than with 32 threads when running 15 tests.

Data chunk size

After comparing the achieved speedups to those mentioned in the original article [3], it seemed that achieved results on both the LOS and the NBP system, differed lot from the expected speedups. There are two main variables in the Phoenix system, which are the number of workers (threads) to use and the data chunk size. The number of threads is one of the most essential variables in this research. Therefore this variable was already being altered from time to time. The data chunk size remained on its default value, so I decided that this variable should be altered from time to time too. For this, another bash file is written, so not only the number of threads could be altered easily, but also the chunk size would be different. To run a batch of tests, *runBatchOfBatches.sh* A.3 is used. This script uses *runBatch.sh* in its execution, so both the scripts should be in the same directory. *runBatchOf-*

Batches.sh requires the following parameters.

Parameter	Value
$1	The maximum power of 2 for the number of threads
$2	The start data chunk size (in bytes)
$3	The maximum data chunk size (in bytes)
$4	The data chunk size step (in bytes)
$5	The program to be executed
$6	The data set to analyze
$7	The number of runs
$8	The prefix of the filename to write the output to
$9	The suffix of the filename to write the output to

With this bash file many runs were executed, resulting in a large number of execution times. The goal was to analyze the data by looking at the differences in execution time by adjusting the chunk size.
The graphs in Figure 3.5, 3.6 and 3.7 show the execution time when adjusting the chunk size for the three data set size for *word_count*.

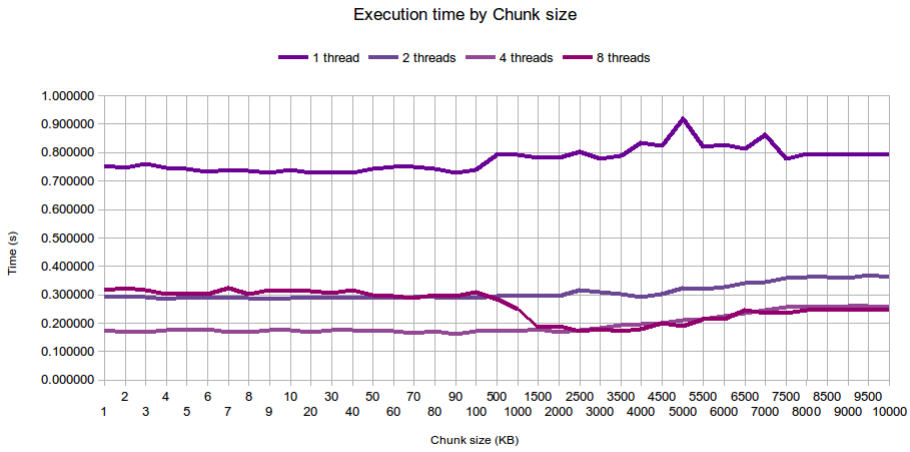

Figure 3.8: Execution time by chunk size for *word_count* 10 MB data set.

These graphs show that the chunk size does not significantly affect the execution time of the algorithm. Most lines perform best with chunk sizes lower than 1 MB, but since Hadoop is built for larger chunks, the applications will both run with chunk sizes of 1 MB.

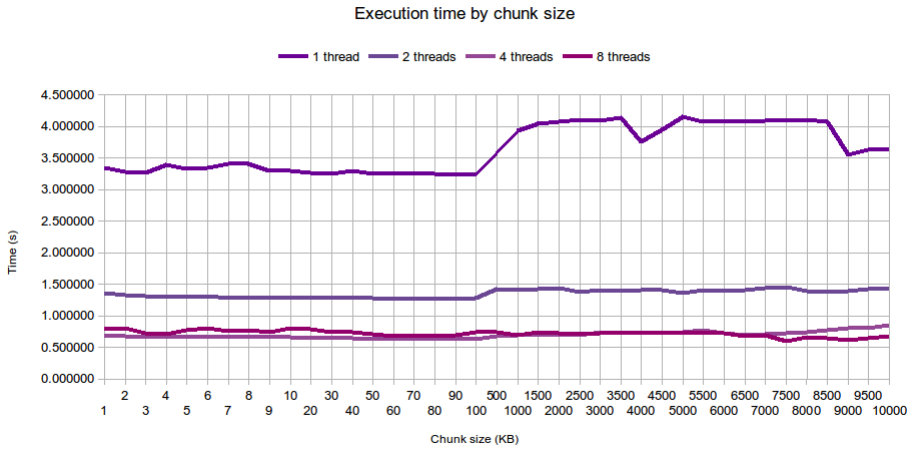

Figure 3.9: Execution time by chunk size for *word_count* 50 MB data set.

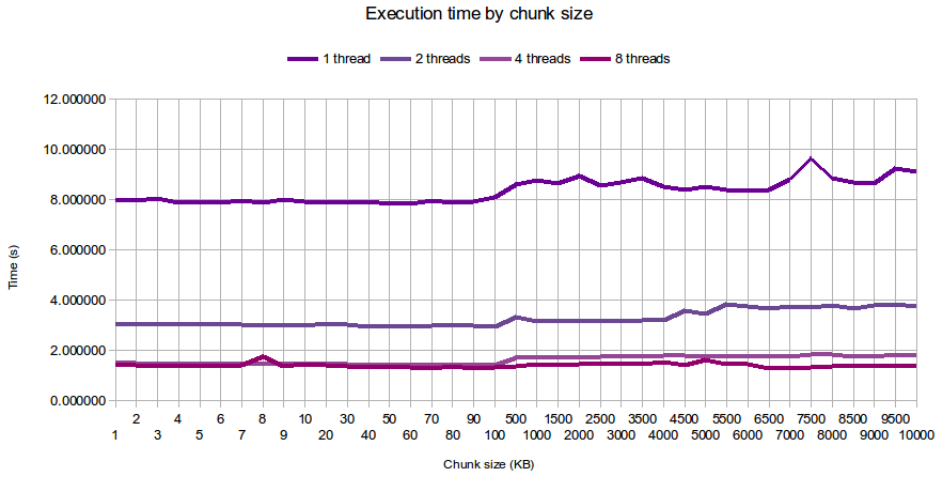

Figure 3.10: Execution time by chunk size for *word_count* 100 MB data set.

3.3 Analysis

The scalability of Phoenix did not appear to get better. Where almost all tests scaled pretty good in the suggested benchmarks, they did not for these experiments. The data chunk size did have a significant influence on the performance of the algorithm and also when measuring different execution times for the same algorithm (e.g. starting time after reading in all data), it did not scale as well as in the initial research paper [3].

However, the Phoenix 2 implementation appears to be known to have issues. In the follow-up article about Phoenix++ [4], the authors show that Phoenix 2 does have scalability issues. By rewriting the whole Phoenix system to C++, they attempt to fix these issues. By redesigning the internal data flows and data storage, they succeeded to fix the scalability issues in Phoenix 2 with Phoenix++. The performance issues I encountered may be solved with this new version of Phoenix.

Another group of researchers has also found a solution for the falling speedup for the Phoenix framework. They built a new implementation based on the original Phoenix, called Ostrich [1]. In this implementation, the authors focussed on optimizing resource usage and they modified the data processing flow. By using a *tiling-strategy*, reusing resources, exploiting data locality and using an NUCA/NUMA-Aware scheduler, Ostrich outperforms Phoenix on almost every field, while using less resources. Although the authors do not speak of Phoenix as a bad scaling framework, they managed to increase its performance significantly.

3.4 Conclusion

To the question whether it would be possible for me to verify the suggested speedups [3], the answer would be no, for now. I successfully managed to redo the experiments and analyze them. Even serveral attempts to verify the suggested speedup results, by changing configuration and scripts, failed. However, there are solutions for the encountered issues suggested in other research papers. All these solutions are new implementations of the Phoenix model that overcome the known disadvantages.

Chapter 4

Hadoop

4.1 Preliminaries

Hadoop is a large software framework developed for distributed computing. The framework consists of many different parts and can be deployed in many different ways. One of the central parts of Hadoop is the Hadoop Distributed File System (HDFS) in which large, distributed files can be stored. Hadoop also contains an implementation of MapReduce. This part will take care of all computations. Hadoop is designed for deployment on large clusters, that contain thousands of nodes. These nodes typically run on many different machines, connected through ethernet. Hadoop has the ability to run in three different modes.

- Standalone
  This is the non-distributed mode of Hadoop. A task (eg. Wordcount) may be assigned to Hadoop and the task will be executed. No installation, HDFS or deployment are required.

- Pseudo-Distributed
  This mode runs on a single machine, The machine itself is the only node in the cluster and is able to handle multiple tasks. This mode must be started and is able to run in the background. Therefor, configuration is required. In this mode, the HDFS is initialized and used for storage.

- Fully-Distributed
  The is the full-scale distributed mode of Hadoop. Many nodes may connect to the Master node through network interfaces. All nodes together also form a large HDFS to store all data.

Because the main goal is to test the Hadoop framework on different numbers of threads on a single system, the Pseudo-Distributed mode is the one that is most suitable for our purpose.

4.1.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [7] is a filesystem that is written in Java and runs on top of the file system that is used by the machine it runs on. The file system is distributed, scalable and portable. It can be used to store large files containing multiple petabytes of data. Therefore the block size is set to 64 MB by default, but many running deployments of Hadoop use block sizes of 128 MB for their file system. Compared to the default block size of NTFS, which is 4 KB, the block size is much larger, so it is more suitable to store large files. Storing small files in HDFS is therefore very inefficient. For every file that is stored on the HDFS, multiple copies are made and stored on other nodes in the cluster. By default three copies are made. These copies are important for reliability and locality. These copies will not be made if Hadoop runs in Pseudo-Distributed mode, since there is only one single node in the cluster.
If a node in the cluster collapses, all the data that were stored on the node is available somewhere else in the cluster and is not lost. Beside that, data that is stored local or very close to all machines, is available for computation faster than when requesting data from a source that is much more further. Therefore, information that is stored on a node, is stored on another node in the same rack as a copy and stored in a node in a different rack as backup.

4.1.2 Hadoop MapReduce

Another main component of Hadoop is Hadoop MapReduce [8], the MapReduce engine. This engine consists of two components: the JobTracker and the TaskTracker. The JobTracker handles the MapReduce requests from the users and delegates the work to be done to available TaskTrackers. The JobTracker also knows which data is stored where, so it can give the work to TaskTrackers close to the data to be handled. When the TaskTracker receives a request, it activates available nodes on the machine by starting them in separate Java Virtual Machines (JVM). These nodes will then process the data and report the results to the TaskTracker. The TaskTracker keeps the nodes busy and reports back to the JobTracker, that can mark a MapReduce request as finished and keeps track of the data. The number of threads that is used by the TaskTracker cannot be set individually, but is defined by the number of active nodes. Each node may be able to use logical threads at the time, but no more than that.

4.1.3 Performance

There are almost no articles that compare performance measurements from Hadoop with a system like Phoenix. This may be so, because the Hadoop framework is developed by many people from different locations all over the world and Phoenix is developed as a research project on a university. Where Phoenix has clear comparison to sequential code, Hadoop does not have comparisons like this. There is one article that compares the performance of Hadoop to Ostrich (a follow-up on Phoenix), but they compare the actual computation time, instead of the scalability of the frameworks. Because the Hadoop framework is so complicated, it is not fair to program a simple sequential program for counting words in Java. Reason for this is because the Hadoop framework does many more things than just counting occurrences of words. The simple sequential algorithm would therefore be much more lightweight than Hadoop itself. Therefore the speedup cannot be computed the same as the Phoenix version of the algorithm.

4.2 Hadoop Performance Comparison

The second goal of the research is to compare two MapReduce frameworks. In this section, a method is given to compare Hadoop to Phoenix, including results.

Like Phoenix, the Hadoop framework is available for download from the website. It is an open source program that runs on all operating systems that run Java. The package contains everything that is needed to run Hadoop on any sort of system or cluster. There are also some examples included. One of these is Word Count, just like Phoenix. This is, however, the only example that is included in both Phoenix and Hadoop. Therefore, this is the only test that will be executed to compare to the Phoenix framework. The data sets that came with Phoenix are used for this experiment.

4.2.1 Research plan

Description

When having unpacked the Hadoop framework, it is ready to run in standalone mode without needing any configuration. Just call the command and the input file is processed. A lot of background actions are displayed in the terminal, but no execution time is provided. Also, a HDFS should be initialized to store the input and output files on. In order to run the Hadoop server in Pseude-Distributed mode, some configuration is required. Within the configuration files that must be set, the number of map-nodes and reduce-nodes per TaskTracker is defined. By modifying these values and restarting the Hadoop server, tests can be run. The same Word Count data sets are used for the tests for Hadoop.

Hadoop will only be executed in the NBP system, because a server needs to be set up and the LOS system does not support setting up such a server.

Speedup measurement

Because there is no sequential algorithm in Java provided with Hadoop, this test cannot be used as reference for other results. Therefor, the first test result, the one with the lowest number of threads possible, would be the reference for the others. The speedup relative to the (expected) slowest configuration is computed with this method. To be able to compare this to the Phoenix results, this method should also be applied to those results. The formula that is used for speedup computation is the following.

$$S_n = \frac{t_1}{t_n} \tag{4.1}$$

Where S_n is the speedup in configuration n, t_n the exectution time for configuration n and t_1 the execution time in the configuration with the lowest number of nodes.

Configuration

Where setting configuration settings for Phoenix was as easy as defining environment variables, the Hadoop framework requires some more configuration for its servers and node settings. The most important variable is the number of map-nodes and reduce-nodes it should use. Because both values need to be 1 or higher, a single-thread setting is not possible. Since small experiments with the server showed that the reduce task was completed almost instantly after all map tasks, I decided to set a maximum number of reduce-nodes to 1. The maximum number of map-nodes would be changed per test from 1 to 7. The minimum threads that are used would than be 2 and the maximum remains 8.

While running tests, there was most of the time only 1 node active at the time, even if the configuration was set to have more active nodes. The reason for this behavior appeared to be the block size of the HDFS on which the data files were stored. This block size was set to 64 MB, so the data sets of 10 and 50 MB did fit into a single block, therefor allowing only one node to work on it. By setting the block size of the HDFS to 1 MB, blocks of 1 MB were created and all data sets could be executed by multiple nodes at the time.

Running the tests

Running tests on Hadoop is a somewhat bigger challenge than running tests on Phoenix, because the server needs to be running and needs restarting on every change in configuration. Because the data sets are stored on the internal solid state drive of the system and solid state drives do not have idling issues like conventional hard drives, these tests are executed just 3

times. The execution time of one task can be read from the web interface and is displayed in seconds. Because the execution times of Hadoop are a lot larger than those of Phoenix, this would not be a problem.
The tests that were executed on Hadoop were pushed to the JobTracker by hand, so no batch scripts or automatically generated CSV files were involved.

4.2.2 Results

Figure 4.1 displays the results of the *word_count* tests that were run with the Hadoop framework. The speedup is very consistent, it never gets slower when more threads are running.

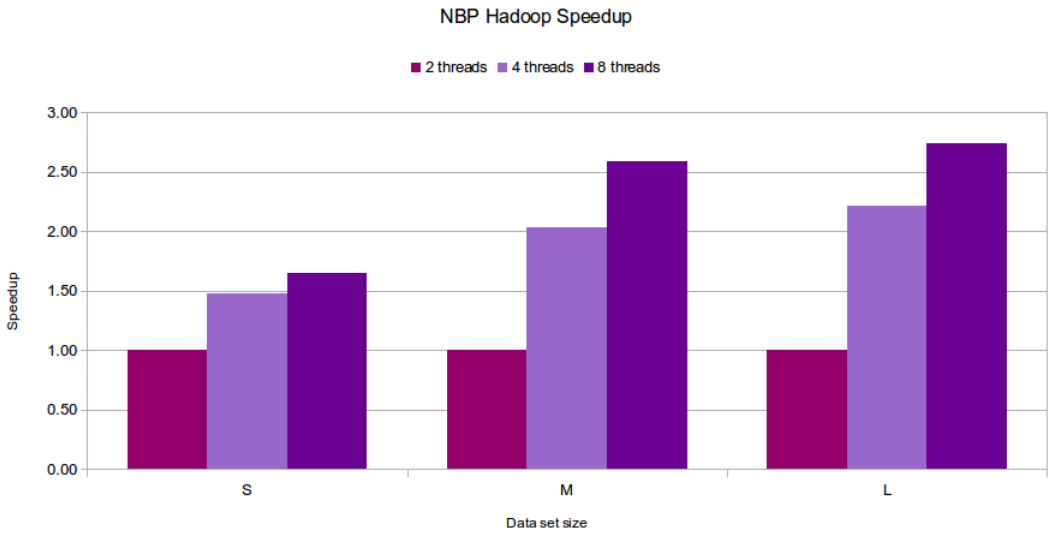

Figure 4.1: The NBP speedup for *word_count* on the Hadoop framework.

4.3 Analysis

When comparing the Hadoop framework with the Phoenix framework, the comparison must fair. Therefore the results of the Phoenix framework for *word_count* must be compared to those of Hadoop. Also the speedup must be calculated using the same formula, so this graph must be made. Figure 4.2 shows the graph for the speedup of the Phoenix system for *word_count* on the NBP system using the speedup formula of Hadoop.

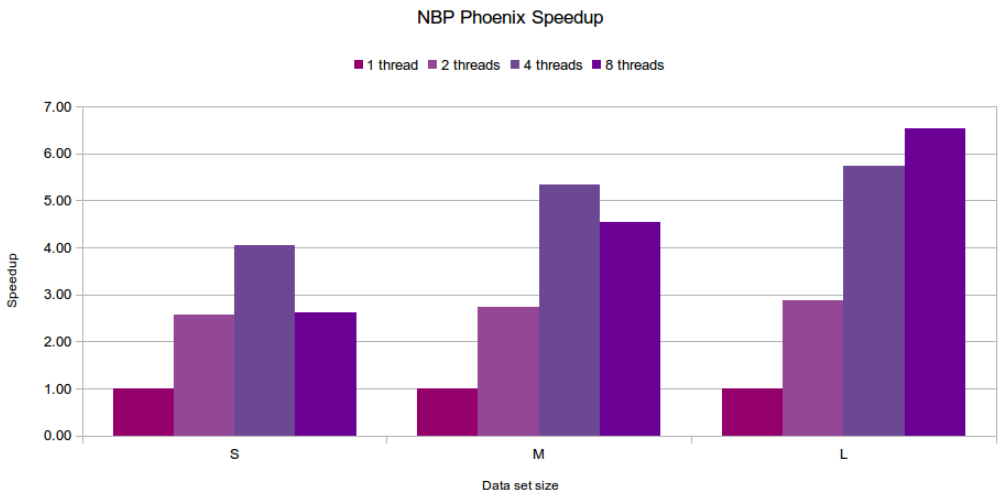

Figure 4.2: NBP speedup for Phoenix *word_count* using the computation of Hadoop.

4.3.1 Speedups

When comparing these results, the main difference is the speedup continuity. With the Phoenix framework two of the three data sets fail to achieve positive speedups for an increasing number of threads, while the Hadoop framework does maintain stable speedups.
Another important notion to make is that the speedups in the Phoenix framework are much higher than in the Hadoop framework.

4.3.2 Memory consumption

An important difference that cannot be read of these graphs is that the memory consumption for Hadoop is much higher. Where Phoenix only has to run an algorithm, Hadoop starts a whole JobTracker for running such a small test. This JobTracker runs in a Java Virtual Machine (JVM) and consumes some resources. On top of that, the JobTracker delegates work to a TaskTracker, which is also running in its own JVM and therefor consumes another amount of memory. The TaskTracker keeps track of the nodes, that are all running in its own JVM. Because of this architecture, the process as a whole consumes much more memory for completing a simple task as counting words. A single JVM could easily consume about 200 MB of working memory. Because of the resource usage of these background processes, it might slow down the system and therefor have lower speedups than the Phoenix framework. Since Hadoop is originally designed for very large files and runs on large clusters, this is not a problem for those applications.

4.3.3 Execution time

One of the main reasons for comparing speedups of two frameworks with each other is because the values are normalized. One can easily compare two frameworks, without looking at the original execution times. These execution times are lost when displaying the speedups, but may be of some importance. The execution time for data sets on Hadoop differed quite some from the execution time for the same data set on Phoenix. Phoenix performed much better. In many tests, Phoenix performed twenty times better than Hadoop. A possible reason for this big difference may be that Hadoop is not very suitable for small-scale setups like this one. Also Hadoop is developed to handle much larger data sets than these small data sets.
As an additional reason for the bad execution times for Hadoop, the fact that Hadoop needs to create multiple objects for the single piece of input, as stated in the article about Ostrich [1]. In this article the performance of Hadoop was compared to the performance of Ostrich, by comparing execution times of both systems. Because Hadoop is much larger than Phoenix, I've chosen another method for comparing them.

4.4 Conclusion

A careful conclusion may be drawn from this comparison. First of all, the performance of Hadoop as measured in these experiments may not be based on the most efficient code for such experiments, while Phoenix was benchmarked with these tests before. Also, the number of tests that is run is too small to draw a solid conclusion. Having these things clear, the scalability of Hadoop was more solid than the scalability of Phoenix. The speedup did not decrease as more threads were added to the computation, as it did for Phoenix. Hadoop however, consumes a lot more memory than Phoenix when in use, due to the JVMs that need to be started.

Concluding from the performed experiments Hadoop is a more solid framework, but Phoenix is faster.

Chapter 5

Related Work

Most of the related work is already mentioned in the chapters above. To understand the MapReduce programming model [2], the first article about MapReduce was used.

Information about Phoenix [3] was taken from the first paper about the framework, in which the authors explained the programming model and showed some performance achievements. These performance achievements were the main goal for this research, therefore this paper is one of the most important papers used.

Articles about Tiled-MapReduce [1] and Phoenix++ [4] were used when searching for possibly overlooked tunables or issues with Phoenix after achieving some speedup results, because suggestions are made in these articles to improve Phoenix.

When comparing Hadoop to Phoenix, not many Phoenix-like speedup comparison methods were used. When introducing Ostrich [1], the authors compared the framework to Hadoop, but they only compared execution times, instead of speedups.

Chapter 6

Conclusions

On the questions whether it would be possible for me to verify the suggested speedups [3], the answer would be no. I managed to redo the experiments, but the measured speedups were not as suggested. Since the Phoenix 2 implementation of MapReduce is known to have some issues regarding its speedup and a follow-up, Phoenix++, is already released these results may not be very surprising.

By looking at the comparison results for Phoenix and Hadoop, a careful conclusion may be drawn: Hadoop has a more linear speedup. Since this conclusion is based on a only few experiments, it is not very solid. Hadoop also has a larger memory footprint than Phoenix and its tests execute much slower than those of Phoenix. More testing is required to draw a solid solution.

6.1 Discussion

Firstly, as mentioned in earlier chapters, scalability issues with Phoenix 2 are known and should already be fixed in a newer version of Phoenix. The conclusion for verifying the suggested results may change when using this new implementation of Phoenix. Secondly, redoing the comparison with Hadoop again wit Phoenix++ may also change this conclusion. As third and last discussion topic, not enough experiments were done for the comparison between the two frameworks. A lot more experiments with different configurations should have been done to draw a solid conclusion. This also is my most important suggestion for future work. More discussion can be found in the *Analysis* sections in the previous chapters.

6.2 Future work

The most important thing that needs to be done in addition to this study is more testing. I've only done a few test runs for each test, with only one test, while this should be a lot more to draw a solid conclusion. In addition to that, the configuration that is used seemed to be more suitable for Phoenix than for Hadoop. One of those settings is the chunk size, which is pretty small for a large system like Hadoop. For a future research it may be useful to compare both frameworks with many different configurations in order to find each others strong and weak spots.

Bibliography

[1] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 523–534, New York, NY, USA, 2010. ACM.

[2] Jeffrey Dean and Sanjay Ghemawat. Map-Reduce: Simplified Data Processing on Large Clusters 0018-9162/95. *D OSDI IEEE*, 2004.

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24, 2007.

[4] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[5] Unknown. hadoop. `http://hadoop.apache.org/`.

[6] Unknown. Hadoop: Powered By. `http://wiki.apache.org/hadoop/PoweredBy`.

[7] Unknown. HDFS User Guide. `http://hadoop.apache.org/docs/stable/hdfs_user_guide.html`.

[8] Unknown. MapReduce Tutorial. `http://hadoop.apache.org/docs/stable/mapred_tutorial.html`.

[9] Unknown. The Phoenix System for MapReduce programming. `http://mapreduce.stanford.edu`.

Appendix A

Appendix

A.1 Test code abstract

All tests were modified to match the following code.

```
1 // Setup all map reduce arguments
2 gettimeofday(&starttime,0);
3 // Execute map reduce functions
4 // Free all variables
5 gettimeofdat(&endtime,0);
6 double spendtime = ((endtime.tv_sec+endtime.tv_usec/1000000.0) -
      (starttime.tv_sec+starttime.tv_usec/1000000.0));
7 printf("%lf\n", spendtime);
```

A.2 Running batches source

The following bash code is used for running tests as batches.

```
1 #!/bin/bash -xv
2
3 for (( i=1; i<=$3; i++ ))
4 do
5         export MR_NUMTHREADS=$4
6         export MR_L1CACHESIZE=$5
7         ./$1 $2
8 done
```

A.3 Running batches of batches source

The following code is used for running tests in multiple different configurations. The number of threads and the cache size change automatically.

```
1  #!/bin/bash -xv
2
3  export MR_NUMTHREADS=1
4  for ((i = 0; i <= $1; i++))
5  do
6          echo "Threads: $MR_NUMTHREADS"
7          for ((c = $2; c <= $3; c += $4))
8          do
9                  echo "Cache: $c"
10                 echo "Running..."
11                 bash runBatch.sh $5 $6 $7 $MR_NUMTHREADS $c >
                       $8_$MR_NUMTHREADS\_$(($c / 1024))KB$9
12         done
13         export MR_NUMTHREADS=$(($MR_NUMTHREADS * 2))
14 done
```